Refresher

When you add a number to a pointer, that number is added, but first it is multiplied by the sizeof the type the pointer points to.

i.e.

```
char *ptr1 = malloc(1);
ptr1 + 1; // adds 1 to pointer
int *ptr2 = malloc(4);
ptr2 + 1; // adds 4 to pointer, because sizeof int is 4
char *ptr3 = malloc(1);
ptr3 + 2; // adds 2 to pointer
int *ptr4 = malloc(8);
ptr4 + 2; // adds 8 to pointer

void *ptr5 = malloc (some_size);
ptr5 + 1; // demons fly out of your nose (a.k.a. invalid)
```

Pointer Arithmetic

question about pointer arithmetic: how does the computer do arithmetic on void * pointers returned by malloc?

standard that we can't do arithmetic on void pointers, so we must cast

best way to do it would be to use char * so we operate at the byte level

what about using unsigned x = (unsigned) malloc…?

works on some, not on others

void * sizes are different because of 64 bit vs 32 bit

the best way to do pointer arithmetic with numbers is using the magical **uintptr_t**

Casting

e.g. for style

```
int f(void) {

    void *x = malloc(1);

    void *y = malloc(2);

    char *z = (char *) malloc(3);

    unsigned asdfasdfasdf = malloc(4);

    // cast to void to avoid unused variable warnings

   (void) x, (void) y, (void) z;

}
```

the only place where the (char *) in front of malloc is completely necessary is in front of the one in unsigned because of the warning (initialization makes int from point without cast)

C Pattern: Cast unused variables as (void) to avoid compiler warnings.

Memset

```
int main(void) {

    int x;

    memset(&x,61,sizeof(int);

    printf(%d\n,x);

    return 0;

}
```

get unexpected result of 1027423549...but in hex it is 3d3d3d3d which is just 61 4 times. if we think about what memset does, this makes sense, because fills the first n bytes of the pointer with the value interpreted as unsigned char.

sizeof (int): numerically, 4. semantically, the number of bytes it takes to hold an integer.

check out lmgtfy.com/?q=memset; alternatively, type “man memset” into your command line to open up the manual page for memset.

Nasal Demons and Other Undefined Behavior

```
int main(void) {

    int x;

    memset(&x,61,61);

    printf(%d\n,x);

    return 0;

}
```

prints value first then stack overflow...why print first? the undefined behavior didn't go off until main tried to return because we are overwriting the main return address.

if we change the last 61 to 6161616161, then we try to write into the kernel, which is forbidden, so there isn't a print, just a stack overflow

Stack Frames

factorial.c

calculates the factorial recursively.

factorial (0) = =1

factorial(n), when n > 0 == n * factorial (n-1)

however in this code, we don't actually check for n > 0, so if we give it a negative number, the computer segfaults

```
#include <stdio.h>

#include <stdlib.h>
```

```
int factorial(int n) {
    int x;
    if (n == 0)
        x = 1;
    else
        x = factorial(n - 1) * n;
    return x;
}

int main(int argc, char **argv) {
    int n = 2;
    if (argc >= 2)
        n = strtol(argv[1], 0, 0);

    printf("%d! == %d\n", n, factorial(n));
}
```

all local variables are stored in the stack, for each local area, the local variables are stored in a stack frame

stack frame: contiguous region of stack memory holding local variables for an active function (in the process of executing)

if we call factorial(3), there are 4 simultaneous stack frames (one for each 3,2,1,0)

how can we verify this?

what if we print the address of the function at each iteration?

printf("%p\n",&function);

0x804844c

this isn't exactly right, because all the addresses are the same. this corresponds to the global variables section.

now let's print the address of the local variable

prinft("%p\n",&n);

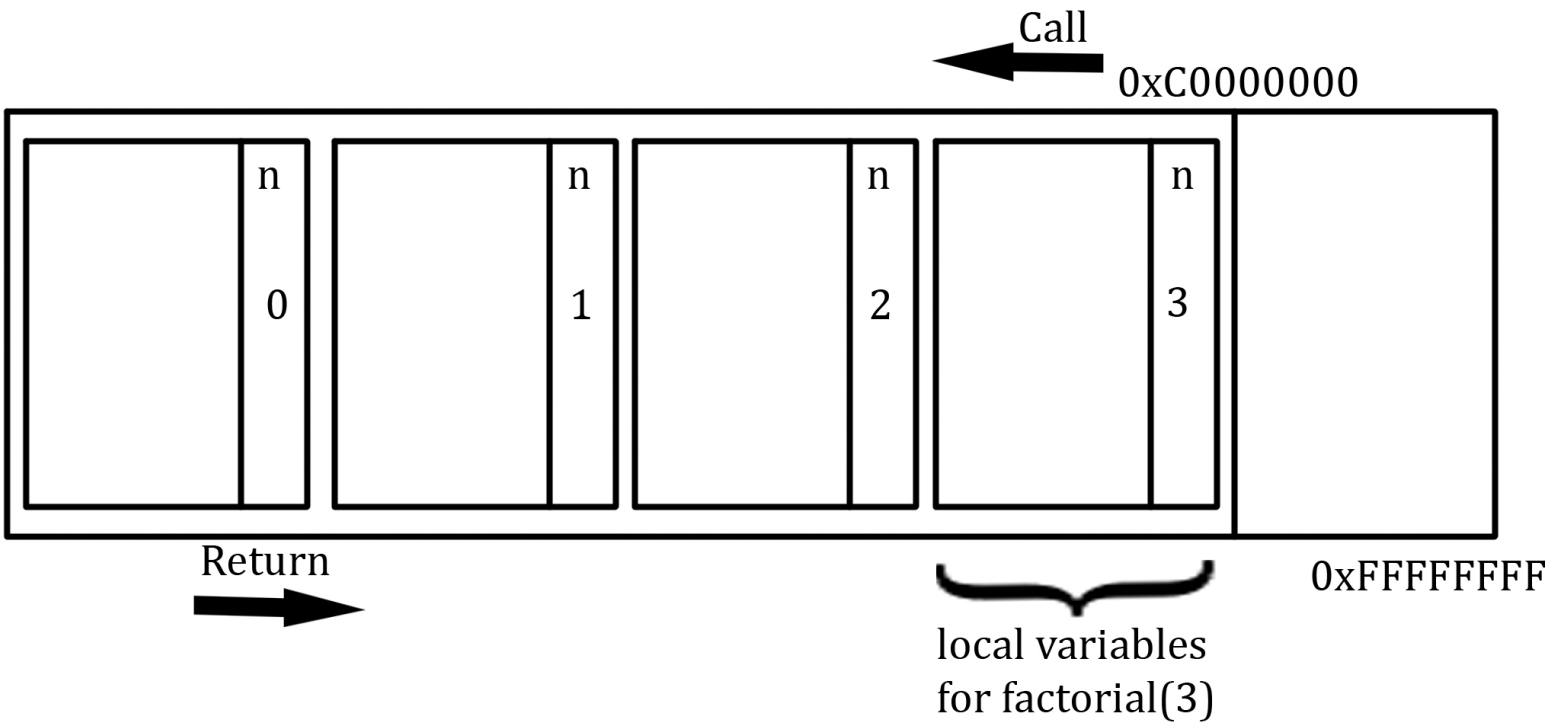
0xbf82a9e0

0xbf82a9b0

0xbf82a980

0xbf82a950

you can 4 print statements, so we can tell that there are 4 stack frames running. these are all spaced 48 (30 in hex) bytes apart. the values also decrease, because the stack grows downwards.



we could also use gdb to do this too

gdb factorial 3

r //runs program; exits without any info

b factorial // sets break at factorial

p n // print n == 3

bt // backtrace; shows active stack frames

up // moves up one stack frame; we can print local vars in this way for each frame

c // continue; continue again until reach n == 0

bt // now can see there are 4 stack frames

GDB Reference

To open: `gdb <program to invoke>`

To set breakpoint `b <function name>`
To run `r <arguments>`
To look at variable `p <variable>`
To step (run the next line only) `s`
To backtrace (print all the current stack frames): `bt`
To continue (run until the next break point): `c`

Note: if you run a program in GDB without arguments, gdb will apply the last arguments you used to run it.

Local Stack Frames

how does the function know which value of n to use?

what if we had something like this:

```
struct stackframe {

    int n;

    int x;

    struct stackframe *caller;

}
```

then the function just had to keep track of which frame was the current frame

struct stackframe *esp;

then we could differentially access the variables using esp->n

these are stored in the most expensive memory of the system because it is the fastest type of memory

there aren't that many of them, and they are effectively global

we can look at registers in gdb using the "info registers" command

In short:

To differentiate local variables in different stack frames, the compiler adds a global that points to the current stack frame, and then the code will look in that stack frame for local variables.

That global variable is in register memory (fastest memory, and there is also very little of it, owned by the processor).

Compiler optimizations

some people are obsessed with making the fastest compilers.

the job of the compiler is not to do exactly what the code says, but to take the code written and turn it into code that has the exact same effects. the compiler is allowed to take a function and turn it into something different.

in our factorial example: the compiler figured out that n=2 is a special case. because it thinks we are going to be calling the function with n=2 most of the time, it just computes the factorial while COMPILING, and return 2 without even calling the function when it is actually run. Also, the compiler is able to make the non-tail recursive factorial function into a tail recursive one.

tail recursive factorial (what the smart compiler actually is doing)

```
int factorial(int n, int acc) {

    if (n == 0)

        return acc;

    return factorial(n - 1,n*acc);

}
```

In the above version of factorial, the function calls factorial (n-1, n*acc) but doesn't do anything to the result before returning it. This allows the program to kill the original stack frame, and only run 1 stack frame of factorial at a time.

Endianness

how are values represented in memory?

char x ='a'; ('a' = 0x61)

another representation of a byte (and it is part of life that a byte means 8 bits)

let's look at int x = 'a';

how will the 0x61 be stored in the int?

big endian

like how we write: big digits are on the left

internet works in big endian

little endian

little digits on the left

c is little endian

(this is only in bytes and doesn't apply to the order of bits in the bytes)

```
int main() {

    int y = 'a';

    int *x = &y;

    hexdump(stdout, &x, sizeof(x));

}
```

we get

bf83eadc = d8 ea 83 bf

arrays are laid out contiguously in memory

this is good because if we think of the cache this means that the cache can give more of the array to the processor when needed